

# METHOD AND APPARATUS FOR DEADLOCK PREVENTION WITH DISTRIBUTED ARBITRATION

## **Cross Reference To Related Application(s)**

This application is a continuation of application Serial Number 09/560,910, filed April 28, 2000, entitled METHOD AND APPARATUS FOR PREVENTING DEADLOCK IN A DISTRIBUTED SHARED MEMORY SYSTEM, which is incorporated herein by reference.

## **Technical Field**

The invention relates to computer processors and memory systems. More particularly, the invention relates to optimizing coherent memory access operations within multiprocessor computer systems having distributed shared memory architectures.

## **Background**

Multiprocessor, or parallel processing, computer systems rely on a plurality of microprocessors to handle computing tasks in parallel to reduce overall execution time. One common implementation of a multiprocessor system is the “single bus architecture, in which a plurality of processors are interconnected through a single bus. However, because of the limited bandwidth of the single bus also limits the number of processors that can be interconnected thereto, recently networked multiprocessor systems have also been developed, which utilize processors or groups of processors connected to one another across an interconnection fabric, e.g., a network, and communicating via “packets” or messages.

Typically, in a networked multiprocessor system includes a plurality of nodes or clusters interconnected via a network. For example, Fig. 1 shows an exemplary networked multiprocessor system **100**, in which a plurality of nodes **102** are interconnected to each other via the interconnection fabric **101**, e.g., a network. By way of an example, only two nodes are shown. However, the networked multiprocessor system **100** may have any number of nodes. Moreover, although, in Fig.1, the interconnection fabric **101** is shown to provide interconnections only between the nodes **102**, all system entities, including the cells **103**, the processors **105** and the memories **104**, are interconnected, and communicate, with the rest of the system through the interconnection fabric **101**.

Each of the nodes **102** of the networked multiprocessor system **100** may be further divided into a smaller hierarchical units — referred herein as “cells” **103**—, which

1 comprises a plurality of processors **105** and a shared memory **104**. Each processor **105**  
2 may comprise any processing elements that may share data within the distributed shared  
3 memory in the system, e.g., a microprocessor, an I/O device or the like. The grouping  
4 into nodes and/or cells of the system entities may be made physically and/or logically.

5 Each of the shared memory **104** may comprise a portion of the shared memory for  
6 the system 100, and may include a memory controller and/or a coherency controller (not  
7 shown) to control memory accesses thereto from various processors in the system, and to  
8 monitor the status of local copies of the memory stored in caches in various processors in  
9 the system using a coherency directory that are maintained in each node or within each  
10 cell. As shown in Fig.2, a typical shared memory **104** receives memory request packets  
11 through the blocking (BL) queue **204**, receives response packets from processors **105**  
12 through the processor return (PR) queue **203**, and sends memory return packets in  
13 response to the memory requests through the MR queue **202**. The PL/BL queue **201** is a  
14 buffer to hold incoming memory requests, where a BL transaction is a transaction  
15 involving a memory access request to the shared memory **104**, e.g., a memory read  
16 transaction, and where a PR transaction is a transaction involving a response from a  
17 processor **105** in response to a coherency check request from a shared memory **104** and/or  
18 a write back of a copy of a cache line in its cache back to the memory **104**.

19 One significant problem that exists with many networked computer systems is that  
20 of *deadlock*, where nodes may in effect “lock up” due to an inability to pass packets or  
21 messages to other nodes. In particular, in some networked computer systems, sending a  
22 primary request to another node may result in the receiving or destination node sending  
23 out one or more secondary requests, e.g., to notify other nodes having local copies of a  
24 memory block that the copies of the memory block in their respective caches must be  
25 marked invalid (often referred to as “invalidation” requests). However, if one or more  
26 secondary requests cannot be sent by the destination node, the node may block receipt of  
27 new requests from other nodes. This may result in two nodes each waiting for responses  
28 from the other.

29 For example, if a shared memory **104** receives a BL transaction, e.g., a memory  
30 read request from a processor, the memory **104** may send a MR request, e.g., a coherency  
31 check request, to other processor(s) to ensure that no copies of the requested data reside in  
32 caches of other processor(s). The shared memory **104** then must wait for processor  
33 responses (PR) from the other processor(s) before it can issue a MR transaction that  
34 satisfies the memory read request. However, if the MR transaction to other processors

could not be issued, e.g., because the MR queue is full at the time, then no other transaction can be received by the shared memory 104, and thus a deadlock occurs.

For optimal performance of the shared memory system, it is crucial that the exchange of packets among the processors and shared memories continuously flow.

Prior attempts to address the deadlock problem includes provision of MR queues which can hold as many memory return transactions as BL transactions in the PR/BL queue may generate. Unfortunately, however, this approach requires a large buffer area. The control logic necessary to control this larger buffer area also becomes very large and complex, and thus is more difficult to design and too slow to operate.

Moreover, because the number of PR transactions and the number of BL transactions that can be processed at any given time are each fixedly arranged and independent with respect to each other, a conventional shared memory system cannot dynamically adapt to process more of BL transactions or more of the PR transactions as the realtime need may require, and is thus inflexible.

Another prior attempt to address the deadlock problem is to provide a special entry type dedicated to handle write-back PR transactions. This approach requires an additional design effort, which is difficult, often "bug prone", and the resulting design is often very inflexible.

Moreover, conventional memory access transaction queues allow only one MR transaction to be added per clock cycle, and thus are inefficient.

Thus, there is a need for more efficient method and device for memory access in a distributed shared memory system, which prevents occurrences of deadlocks, which does not require a large MR queue, and which does not require a design of a special dedicated entries for write back transactions.

There is also a need for more flexible method and device for memory access in a distributed shared memory system that may be dynamically adaptive to the current requirement for processing various memory access transactions.

There is also a need for a distributed queuing mechanism that allows multiple addition of entries in a clock cycle.

## **Summary**

In accordance with the principles of the present invention, a method of preventing a deadlock in a distributed shared memory system having a memory access request transaction queue having a plurality of queue slots includes reserving one or more queue slots for exclusive processing of processor return flow control class transactions.

1 In addition, in accordance with the principles of the present invention, an  
2 apparatus for preventing a deadlock in a distributed shared memory system having a  
3 memory access request transaction queue having a plurality of queue slots includes a  
4 coherency controller configured to reserve one or more queue slots for exclusive  
5 processing of processor return flow control class transactions.

#### 6 **Description of the Drawings**

7 Features and advantages of the present invention will become apparent to those  
8 skilled in the art from the following description with reference to the drawings, in which:

9 Figure 1 is a block diagram of the relevant portions of an exemplary conventional  
10 networked multiprocessor system;

11 Figure 2 is a block diagram of the relevant portions of an exemplary conventional  
12 memory access transaction queue system;

13 Figure 3 is a block diagram of the relevant portions of an embodiment of the  
14 memory access transaction queue mechanism in a distributed shared memory system in  
15 accordance with the principles of the present invention;

16 Figure 4 is an illustrative flow diagram showing an exemplary embodiment of the  
17 memory access transaction queue process in accordance with the principles of the present  
18 invention;

19 Figure 5 is an illustrative logic diagram showing an exemplary embodiment of the  
20 circuit for generating global addition signals in accordance with the principles of the  
21 present invention;

22 Figure 6 is a block diagram showing input and output signals of an exemplary  
23 embodiment of the global queue position logic in accordance with the principles of the  
24 present invention; and

25 Figure 7 is a block diagram showing input and output signals of an exemplary  
26 embodiment of the entry queue position logic in accordance with the principles of the  
27 present invention.

#### 28 **Detailed Description**

29 For simplicity and illustrative purposes, the principles of the present invention are  
30 described by referring mainly to an exemplar embodiment thereof. However, one of  
31 ordinary skill in the art would readily recognize that the same principles are equally  
32 applicable to, and can be implemented in, a multiprocessor shared memory system having  
33 a different implementation or architecture, and that any such variation would be within

1 such modifications that do not depart from the true spirit and scope of the present  
2 invention.

3 For example, while much of the following description of the present invention  
4 makes references to multiprocessor systems, it should be appreciated that the concept of  
5 distributing tasks between processors in multiprocessor systems may also be applied to  
6 distributed computer systems which distribute tasks between different computers in a  
7 networked environment (e.g., a LAN or WAN). Further, many of the functions and  
8 problems associated with multiprocessor and distributed computer systems are quite  
9 similar and equally applicable to both types of systems. Consequently, the term  
10 “networked computer system” will be used hereinafter to describe both systems in which  
11 the nodes are implemented as individual microprocessors or groups of processors  
12 (multiprocessor systems) or as individual computers which may separately utilize one or  
13 more processors (distributed computer systems).

14 In accordance with the principles of the present invention, a distributed shared  
15 memory system having a memory access request transaction queue having a plurality of  
16 queue slots prevents occurrences of deadlocks. The distributed shared memory system is  
17 implemented in a networked multiprocessor computing system, and includes, in each  
18 coherency controller of each of the memories in the system, a mechanism to reserve at  
19 least one slot of the memory access request transaction queue for exclusive processing of  
20 processor return (PR) transactions to provide an uninterrupted processing of PR  
21 transactions. The number of blocking (BL) transaction is limited to a number less than  
22 available slots.

23 The inventive distributed shared memory system also includes a distributed  
24 memory return transaction queue that allows each of entries in the memory access request  
25 transaction queue to add a plurality of memory return transactions per clock cycle.

26 In accordance with the principles of the present invention, the coherency system  
27 for the networked multiprocessor system divides the packet transactions traffic into, *inter*  
28 *alia*, three general flow control classes as shown in table 1:

29 Table 1 Flow control class dependencies

Class	Can Generate
Blocking (BL)	Memory Return (MR)
Memory Return (MR)	Processor Return (PR)
Processor Return (PR)	Memory Return (MR)

1       A BL transaction is a transaction involving a memory access request to the shared  
2 memory **104**, e.g., a memory read transaction, and where a PR transaction is a transaction  
3 involving a response from a processor **105** in response to a coherency check request from  
4 a shared memory **104** and/or a write back of a copy of a cache line in its cache back to the  
5 memory. As shown in the table, the flow control classes have dependency relationships,  
6 in which a BL transaction can generate another MR transaction, which may in turn  
7 generate a PR transaction, which can also generate yet another MR transaction. This can  
8 be shown pictorially as BL -> MR -> PR (-> MR), where “->” is read “can generate”.

9       In accordance with the principles of the present invention, a deadlock is prevented  
10 by ensuring that the PR flow control class never be blocked. Since accesses to the shared  
11 memory by the coherency controller are dependent only upon availability of sufficient  
12 memory bandwidth, write-back transactions can be processed as long as there are  
13 available slots in the memory access transaction queue, i.e., the PR/BL queue **201** (shown  
14 in Fig. 3). The PR/BL queue **201** is a fixed length buffer that has n number of slots. In a  
15 preferred embodiment of the present invention, the PR/BL queue **201** comprises 28 slots,  
16 and thus may process only 28 transactions of either BL or PR flow control class at any  
17 given time.

18       When the PR/BL queue **201** is filled up, no other transactions can be received, and  
19 thus the PR/BL queue **201** must be stalled. In accordance with the principles of the  
20 present invention, to ensure that PR write-back transactions can be processed, at least one  
21 of the PR/BL queue slot must be reserved for exclusive processing of the PR flow control  
22 class transactions.

23       In particular, Fig. 3 shows an embodiment of the inventive memory access  
24 transaction queue system **300**, which comprises a shared memory **104** that includes a  
25 coherency controller **301**, MR queue **202**, multiplexer **302** and a coherency controller  
26 **301**. The multiplexer **302** receives a BL transaction input **307** and a PR transaction input  
27 **308**, and selects one of the two inputs based on a control signal received from the  
28 coherency controller **301**. BL transactions **307** and PR transactions **308** may be inputted  
29 to the multiplexer **302** through respective input buffers, e.g., similar to the BL queue **204**  
30 and the PR queue **203** shown in Fig. 2, respectively. The coherency controller **301**  
31 includes a PR/BL queue **201** that receives the transactions output from the multiplexer  
32 **302** and four registers **303**, **304**, **305** and **306**, in which are stored the “Entry\_Count”,  
33 “BL\_Count”, “Entry\_Threshold” and “BL\_Threshold”, respectively.

1       The “Entry\_Count” register **303** holds a number indicative of the number of  
2 entries of all types that are currently in the PR/BL queue **201**. The “BL\_Count” register  
3 **304** hold a number indicative of the number of entries containing BL transactions  
4 currently in the PR/BL queue **201**. The “Entry\_Threshold” register **305** and the  
5 “BL\_Threshold” register **306** hold the maximum number of entries allowed to be  
6 processed and the maximum number of entries containing BL transaction allowed to be  
7 processed in the PR/BL queue **201**, respectively, both thresholds of which are  
8 configurable by a user of the networked multiprocessor system and/or by system  
9 software.

10       The Entry\_Threshold is selected to be less than the number of slots available in  
11 the PR/BL queue **201**, e.g., 28 in the preferred embodiment of the present invention. The  
12 BL\_Threshold in turn is selected to be less than the Entry\_Threshold.

13       The coherency controller **301** asserts an appropriate signal(s) to the multiplexer  
14 **302** to pass a PR transaction to the PR/BL queue **201** as long as the Entry\_Count is less  
15 than the Entry\_Threshold, and to pass a BL transaction if the BL\_Count is less than the  
16 BL\_Threshold *and* the Entry\_count is less than the Entry\_threshold. Thus, at least one  
17 PR/BL queue slot may be reserved for PR transactions at any given time.

18       In particular, Fig. 4 shows a flow diagram describing the operations of the  
19 coherency controller **301**. In step **401**, one or more new transaction of either PR or BL  
20 flow control class is received at the input(s) of the multiplexer **302**. In step **402**, the  
21 coherency controller **301** examines the current Entry\_Count, and compares the same with  
22 the Entry\_Threshold to determine if the Entry\_Count is less than the Entry\_Threshold.

23       If the Entry\_Count is not less than the Entry\_Threshold, e.g., if they are equal,  
24 then the PR/BL queue **201** is made to stall, i.e., no more new transactions are accepted by  
25 the queue, in step **406**, until one or more entries are processed and retires from the PR/BL  
26 queue **201**, and new slots are freed up, i.e., the Entry\_Count becomes less than the  
27 Entry\_Threshold as shown.

28       If , on the other hand, the current Entry\_Count is less than the Entry\_Threshold,  
29 and if at least one of the one or more new transactions received is a PR transaction, the  
30 coherency controller **301** sends, in step **403**, a signal to the multiplexer **302** to allow the  
31 PR transaction to pass through to the PR/BL queue **201**, and the PR transaction is  
32 accepted.

33       In step **404**, if at least one of the one or more new transactions received is a BL  
34 transaction, the coherency controller **301** examines the current BL\_Count, and compares

1 the same with the BL\_Threshold to determine if the BL\_Count is less than the  
2 BL\_Threshold.

3 If the BL\_Count is not less than the BL\_Threshold, e.g., if they are equal, then the  
4 PR/BL queue 201 is made to block any new BL transactions are accepted by the queue, in  
5 step 407, until one or more BL transactions are processed and retires from the PR/BL  
6 queue 201 so that the BL\_Count once again becomes less than the BL\_Threshold as  
7 shown.

8 If, on the other hand, the current BL\_Count is less than the BL\_Threshold, the  
9 coherency controller 301 sends, in step 405, a signal to the multiplexer 302 to allow the  
10 BL transaction to pass through to the PR/BL queue 201, and the BL transaction is  
11 accepted. The entire process is repeated when a new transaction is received at the  
12 multiplexer 302.

13 As can be appreciated, the inventive memory access request transaction queue and  
14 the coherency controller in the foregoing description provides a distributed shared  
15 memory system, in which occurrences of deadlocks are prevented without the need for a  
16 large MR queue or a design of a special dedicated entries for write back transactions.

17 In addition to write-back transactions, the PR/BL queue may contain responses to  
18 recalls issued as a result of BL transactions in the queue. Processing of a recall response  
19 for an entry may cause that entry to generate a MR data return. Therefore, a MR data  
20 return may need to be queued up for as many as the BL\_Threshold. Moreover, MR data  
21 returns may also result from, *inter alia*, directory tags and data returning from the  
22 memory, BL transaction linked list advances, and timeouts. All of these MR data returns  
23 may occur during the same clock cycle. Thus, up to four additions could be made to the  
24 MR queue in each clock cycle. The MR queue may advance in any clock cycle,  
25 removing the oldest entry from the queue, i.e., first-in-first-out (FIFO).

26 In accordance with the principles of the present invention, the above described  
27 MR queuing needs are met by a distributed queue mechanism, in which up to four (4)  
28 additions and zero (0) or one (1) deletions are allowed to be made during a clock cycle.  
29 The state components of an embodiment of the distributed queue mechanism are two  
30 pointers, global\_queue\_pos and entry\_queue\_pos, both of which are implemented as  
31 seven (7) bit counters in the preferred embodiment of the present invention.

32 One of the two counters is referred to herein as the "entry\_queue\_pos counter",  
33 and is provided for each of the entries that may cause a MR data return. Each of the  
34 entry\_queue\_pos counters outputs a count value, entry\_queue\_pos, the two's complement



1 of which represents the position of the associated entry on the distributed queue. A value  
2 of zero of the entry\_queue\_pos indicates that the entry is the oldest entry in the  
3 distributed queue, or is located at the front of the distributed queue. A negative value of  
4 the entry\_queue\_pos indicates the entry is not in the distributed queue. A positive value  
5 of the entry\_queue\_pos indicate that the entry is in the distributed queue at a location  
6 other than the front of the queue, a larger the positive count value indicates that the entry  
7 is positioned further back in the distributed queue.

8 The other of the two counters is referred to herein as the “global\_queue\_pos  
9 counter”, and outputs a count value, global\_queue\_pos, which represents the position of  
10 the highest numbered, i.e., the newest, entry in the distributed queue. A negative value of  
11 global\_queue\_pos indicates no entries are in the distributed queue. A value of 0 indicates  
12 one entry on the distributed queue. Whenever a new value is updated, i.e., counted up or  
13 down, the updated value of the global\_queue\_pos is broadcast to all entries.

14 In this exemplary embodiment, each entry drives four add lines, add[3:0],  
15 corresponding to the four types of events that may cause simultaneous adds as previously  
16 described. These add lines have a fixed priority order associated thereto, with add[3]  
17 being the highest priority and add [0] being the lowest.

18 As shown in Fig. 5, each of the add lines add[3] 502 from all entries, entry 1 to  
19 entry n 501, are logically ORed together with an OR gate 506 to produce the Global\_Add  
20 [3] 510, each of the add lines add[2] 503 from all entries are ORed together with an OR  
21 gate 507 to produce the Global\_Add [2] 511, each of the add lines add[1] 504 from all  
22 entries are ORed together with an OR gate 508 to produce the Global\_Add [1] 512, each  
23 of the add lines add[0] 505 from all entries are ORed together with an OR gate 509 to  
24 produce the Global\_Add [0] 513. In an embodiment of the present invention, a queue  
25 advance signal is provided to indicate that the distributed queue is advancing,

26 In an embodiment of the present invention, a global queue position logic as shown  
27 in Fig. 6 is provided to recalculate the value of the global\_queue\_pos during each clock  
28 cycle. In particular, as shown in Fig. 6, the global queue position logic 601 receives the  
29 Global\_Add [3:0] 510-513, the queue advance signal 602 and the seven bit current value  
30 of the global\_queue\_pos 603 from the global\_queue\_pos counter. The global queue  
31 position logic adds the number of additions as reflected in the received Global\_Add [3:0]  
32 510-513 to determine the number of newly added MR transactions. If the queue advance  
33 signal is active, i.e., if the distributed queue is advancing, the global queue position logic  
34 subtracts 1 from the number of newly added MR transactions. If the queue advance

1 signal is inactive, no subtraction from the number of newly added MR transactions is  
2 made.

3 The resulting number of newly added MR transactions, i.e., after taking the queue  
4 advance signal into account, is then added to the current `global_queue_pos` **603** to arrive  
5 at the new `global_queue_pos` value **604**, which is stored in the `global_queue_pos` counter,  
6 and is also broadcasted to each of the entries.

7 Additionally, during each clock cycle, location of each of the entries in the  
8 distributed queue is recalculated in its associated entry queue position logic, and  
9 exemplary implementation of which is shown in Fig. 7. An entry queue position logic  
10 **701** is provided for each of the entries. For each entry, during each clock cycle, the  
11 associated entry queue position logic **701** receives the `Global_Add` [3:1] **510-512**, the  
12 queue advance signal **602**, the seven bit current value of the `entry_queue_pos` **702** from  
13 the `global_queue_pos` counter, the current `global_queue_pos` value **603** and the four add  
14 lines associated with the entry, `Add` [3:0] **703**, i.e., addition lines **502-505** shown in Fig.  
15 5, and outputs a seven bit new `entry_queue_pos` value **704** to be stored in the  
16 `entry_queue_pos` counter.

17 The calculation of the new `entry_queue_pos` **704** depends on whether or not the  
18 entry is already on the distributed queue. If, based on the current `entry_queue_pos` value  
19 **702**, the entry is already on the queue, the entry queue position logic decrements the  
20 entry's queue position by 1 if the queue advance signal **602** is active, i.e., the distributed  
21 queue is advancing, and maintains the current position of the entry if the queue advance  
22 signal **602** is inactive.

23 If, on the other hand, the entry is not already in the distributed queue, the current  
24 negative `entry_queue_pos` value is maintained if no additions are being made during the  
25 current clock cycle. If the entry is not already in the distributed queue, but the entry is  
26 adding new MR transactions, the entry queue position logic **701** calculates the new  
27 `entry_queue_pos` value **704** as follows:

28 The entry queue position logic **701** adds the value of all `Global_Add`[3:1] **510-512**  
29 with a higher priority than the `Add`[3:0] **703** to arrive at an add value. If the queue  
30 advance signal **602** is active, the entry queue position logic **701** subtracts 1 from the add  
31 value. If the resulting add value is - 1, then the `global_queue_pos` **604** is the next  
32 `entry_queue_pos` **704**. If the resulting add value is 0 or greater, then the next `entry_queue_pos`  
33 **704** is chosen to be the `global_queue_pos` **604** plus (the add value +1), i.e.,  
34  $\{(\text{global\_que\_pos}) + (\text{add value} + 1)\}$ .

1           As can be appreciated, the distributed queue mechanism in the foregoing  
2 description allows up to four new MR transactions to added by each entry during a clock  
3 cycle.  
4 While the invention has been described with reference to the exemplary embodiments  
5 thereof, those skilled in the art will be able to make various modifications to the described  
6 embodiments of the invention without departing from the true spirit and scope of the  
7 invention. The terms and descriptions used herein are set forth by way of illustration only  
8 and are not meant as limitations. In particular, although the method of the present  
9 invention has been described by examples, the steps of the method may be performed in a  
10 different order than illustrated or simultaneously. Those skilled in the art will recognize  
11 that these and other variations are possible within the spirit and scope of the invention as  
12 defined in the following claims and their equivalents.